Parametric polymorphism, also known as *generics*, is a programming language feature with implications for compiler design. The word polymorphism in general means that there are value or perhaps other entities that can have more than one type. We have already talked about subtype polymorphism, in which one type can act like another type. Subtyping places constraints on how we implemented objects. In parametric polymorphism, the ability for something to have more than one shape is by introducing a parameter that is a type.

A typical motivation for parametric polymorphism is support for collection libraries such as the Java Collection Framework. Prior to the introduction of parametric polymorphism, all that code could know about the contents of a `Set` or `Map` was that it contained `Objects`. This led to code that was clumsy and not type-safe. With parametric polymorphism, we can apply a parameterized type such as `Map` to particular types: a `Map⟨String, Point⟩` maps `Strings` to `Points`. We can also parameterize procedures (functions, methods) with respect to types. Using Xi-like syntax less clumsy than Java's, we might write a `is_member` method that can look up elements in a map:

```
contains⟨K, V⟩ (c: Map⟨K, V⟩, k: K): Value { ... }
Map⟨K, V⟩ m
...
p: Point = contains⟨String, Point⟩(m, "Hello")
```

Although `Map` is sometimes called a *parameterized type* or a *generic type* , it isn't really a type; it is a type-level *function* that maps a pair of types to a new type. We might denote its signature as `type*type→type`. This type-level function is evaluated at compile time rather than at run time, by applying it to type arguments. The result of such an application is an *instantiation* of the generic type or function.

Parameter types From the perspective of building a compiler for a language, a big change introduced by parametric polymorphism is the introduction of a new kind of type, the *parameter type*. Inside the definitions of `Map` or `contains`, we can use the names `K` and `V` to refer to the types on which these generic abstractions have been instantiated.

When inspecting the generic code, we don't know what argument types these parameter types stand for. Therefore, if code is to be generated for the generic code, it must be prepared to handle a value of any type that is usable as an argument.

One challenge of generating generic code is types whose representations have different sizes. For example, in Java we cannot use primitive types as type parameters because types such as `long` do not fit into a word.[1]

Some languages support *constrained parametric polymorphism* in which constraints can be placed on type parameters. The constraint forms a contract between clients using the parameterized abstraction, and the code that implements the abstraction. The compiler can then make use of this information both when type-checking code that uses a type `T` and when generating code for it.

1 Templates

The approach taken by C++ is to check generic code *after* it has been instantiated. Parameter types are not explicitly constrained, and it is legal to use them in any way that is desired. When a parameterized class or function is instantiated, the actual type arguments are substituted for the occurrences of the type parameters, and only then does type checking and code generation occur.

This approach is relatively simple and has some advantages. Since each separate instantiation has, in general, its own code, the generated code is automatically specialized to the particular types being used, making it as efficient as we could expect. It is also possible to use types of different sizes and kinds.

[1] Limitations of the Java Virtual Machine are also partly to blame

Instantiating on primitive types like `char` (8 bits), `int` (32 bits) or `long long int` (64 bits) is allowed. If a parameter type *T* is instantiated with, say, `char`, an array of type *T*`[]` is a real, efficient `char[]` array.

The downside of the C++ mechanism is that it is not modular. Generic libraries are distributed in source-code form, so that each instantiation can be checked. Generating code for each distinct instantiation tends to lead to "code bloat" because the different instantiations are largely replicas of each other. (Some modern C++ compilers reduce code bloat by combining instantiations whose code turns out to be identical.) Perhaps the worst problem is that when instantiation fails, the resulting type errors are reported in the context of the generic implementation. The programmer using a generic library is then exposed to details of the implementation that they are unprepared to interpret. Debugging can be very challenging.

The C++ template mechanism is very powerful; it has evolves into a Turing-complete mechanism for compile-time evaluation, and this power has its uses. Unfortunately, the simple things are not so well supported.

As a side note, C++ introduced the syntax of "angle brackets" (<>) to express generics, whereas the typical syntax in earlier languages was square brackets (`[]`). Because the "angle bracket" characters are also used to express comparison and shift operators in C++ and Java, parsing these languages becomes more complex; this is one reason the C++ grammar is ambiguous.

2 Unconstrained polymorphism

Some languages provide mechanisms for unconstrained polymorphism that preserve separate compilation and modularity. In particular, languages from the ML family, such as SML/NJ and OCaml, allow polymorphic functions and types to be defined. For example, in OCaml we can write:

```
type 'a list = Nil | Cons of 'a * 'a list
let rest (lst : 'a list) : 'a list =
    match lst with Nil -> Nil | Cons(h,t) -> t
```

This polymorphism mechanism doesn't allow constraints to be expressed on parameter types, so code cannot be written with the assumption that an argument type has an operation to be used. Any operations that are to be performed on a parameter type must be supplied as a separate first-class function value.

Since few assumptions are being made about the parameter type, it is relatively straightforward for the compiler to implement a *homogeneous translation* in which code for the generic abstraction is generated only once. Code bloat is avoided.

Homogeneous translation has a downside, of course. The code must be able to treat all types in a uniform way. Typically this is done by assuming that all types have a one-word representation. Primitive types that don't fit into a word are represented as a pointer to a *boxed* representation as an on-heap object. This representation can be a high price to pay, especially for arrays containing a parameter type. What would in C++ be an efficiently packed contiguous sequence of elements turns into a large number of objects that the garbage collector must manage.

3 Subtype-constrained polymorphism

A popular alternative strategy for constraining types is to use the existing language mechanism for subtyping. For example, in Java we can declare that the elements of a set must support a comparison operation by requiring that they implement a constraint interface all of whose subtypes have the required `compareTo` method:

```
class Set⟨T extends Comparable⟨T⟩⟩ {
    ...
    T[] elements;
    T x = elements[i];
    ...
    x.compareTo(elements[j]);
}
interface Comparable⟨T⟩ {
    int compareTo(T y);
}
```

With subtype constraints, the code of `Set` can be type-checked in a modular way and a type-safe homogeneous translation is possible, based on *type erasure*. We can express this translation simply a a translation to Java without generics. All parameters are erased and parameter types are replaced with their constraints interfaces:

```
class Set {
    ...
    Comparable[] elements;
    Comparable x = elements[i];
    ...
    x.compareTo(elements[j]);
}
interface Comparable {
    int compareTo(Object y);
}
```

If a constraint interface has a method that returns a parameter type, the type system of a non-generic target language cannot prove the operation is type-safe. Consequently, the Java compiler inserts a run-time cast to convince the JVM that the types match up. For example, code accessing a generic list is translated along the following lines:

Source:

```
class List<E> {
    E get(int index) { ... }
}

List<String> lst = ...
String x = lst.get(0);
```

Target:

```
class List {
    Object get(int index) { ... }
}
List lst = ...
String x = (String)lst.get(0);
```

Assuming the Java type system is sound (which is currently isn't!) the type cast is guaranteed to succeed, and a sufficiently clever compiler could avoid the overhead of the cast.

The Java translation has the same problem as OCaml, that primitive types must be boxed, and this boxing is unfortunately quite explicit in the language, with types like `Integer` and `Long` distinct from `int` and `long`. Boxing can be avoided by generating specialized implementations just for types that would otherwise be boxed. This is the approach taken by C# — a little code duplication may be worthwhile if it results in more efficient code for primitives.

As a language features, subtype constraints do have a serious limitation. An type can only be used as a type argument if it obeys the subtyping rules of the language. In languages like Java and C# where subtyping exists only when explicitly declared—a rule that makes dispatching easier to implement efficiently—the types used as argument have to have been developed with their use as an argument planned ahead of time. This is a real limitation if those types are coming from a library over which the programmer has no control. The typical workaround is to use the Adapter design pattern, wrapping the object in a new object that does declare it implements the constraint interface. But this is an expensive workaround, and especially problematic when code generic on `T` wants to use an array of `T`'s, meaning that many wrappers have to be created.

4 Type class constraints

A solution to the rigidity of subtype constraints is to introduce a separate, more flexible type constraint mechanism. This is the approach pioneered by Haskell with its *type classes*, though some earlier languages, notably Argus, have features heading in this direction. Type classes also correspond to the Concept design pattern developed around the same time in the C++ community (but which has still not made it into the language). Scala supports the Concept design pattern in a lightweight way through its language feature of implicit arguments; the recently developed Genus language also offers a type constraint similar to type classes.

The idea of these mechanisms is that constraints should be expressible on types, and that programmers can then write code (*instances* in Haskell, *models* in Genus and in the Concept design pattern) that provides the necessary glue for the type to satisfy the constraint. Rather that requiring that a type declare it implements a constraint interface, as in Java, a model can in principle be used to adapt *any* type to any constraint. This *retroactive modeling* capability is powerful and useful.

Translating the `Set` example above into Genus, we express the constraint `Comparable` as a predicate on the type `T`:

```
constraint Comparable[T] {
    int compareTo(T x);
}
```

Now, suppose that we want a set of integers, `Set[int]`. It happens that this is legal in Genus without writing more code, because the type `int` induces a *natural model* for the constraint `Comparable`. If `int` didn't do this, or if we wanted to satisfy the constraint in a different way than the default, we can declare a model that describes a different way to satisfy it:

```
model IntComp for Comparable[int] {
    int compareTo(int x) {
        return this - x;
    }
}
```

The type `Set[int with IntComp]` now defines a set of integers ordered according to the `IntComp` model.

Models must be at least implicitly available at run time because they are used to dispatch operations to the right code. This means that models also make it easy to implement *reified generics* in which there is a run-time representation of type arguments. For example, with reified generics we could use `instanceof` and run-time casts to discover the types of objects in a way that type erasure makes impossible in Java:

```
Object o = ...;
Set[int] s = (Set[int]) o;
```

One way to translate type-class generics is to use the concept design pattern, corresponding to the way that they would be programmed in Scala. Models are implemented as classes and constraints become interfaces. Objects of generic classes are augmented fields that contain the models to be used when performing operations on parameter types, and these fields are initialized when the object is being constructed.

```
interface Comparable {
    int compareTo(Object this, Object, x);
}

class Set {
    Comparable T_model;
    ...
    Object elements; // a T[]
    Object x = T_model.array_get(elements, i);
    ...
    T_model.compareTo(x, y);
    ...
    Set(Comparable m) {
        T_model = m;
        ...
    }
}
```

Note that in the translation shown, an array of type `T[]` is implemented as a real array of `T`, rather than requiring boxing for array elements in the case where `T` is primitive. This more efficient representation comes at the price of requiring that array operations be dispatched via the model.

It is also possible to specialize generic classes to particular types, avoiding dispatch through the model and even the need to represent the model. To allow some sharing between specialized implementations, model operations can be dispatched through the class's dispatch vector rather than through an explicit model object. As usual, specialization trades off code space for more efficient individual implementations.